CSCI662 HW1: Linear Classifiers

Divya Choudhary
MS in Computer Science at University of Southern California/Los Angeles, CA, US
dchoudha@usc.edu

Abstract

Text classification is one of most common tasks in the field of natural language processing. It had solution for important practical problems like spam vs ham classification, information retrieval and others. The complexity of text data has been increasing with the constant growth of digital platforms, new lingo, writing styles etc. This necessitates a much deeper understanding of both the data and models in order to be able to classify these complex text correctly. Selection of the model to be used for a particular task is the most trivial and complex task any machine learning project.This document, *Comparative Study of Linear Text Classifiers* provides a comparative study of multiple linear classifiers across different data sets. The report dives into aspects of general Machine Learning algorithm building steps, feature engineering, nuances of textual data, model's data dependence and a comparison report of results of multiple linear classifiers. The document report has been compiled with comparison between 3 models- Naive Bayes, Perceptron, Logistic Regression.

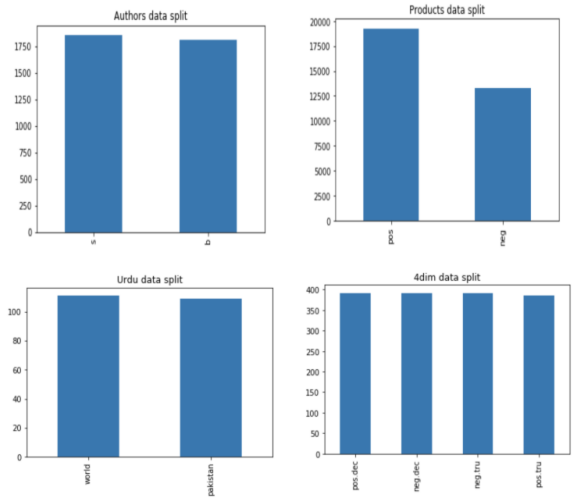
1 Problem Statement

Text classification is an important field of text mining. In a classical , text classification, a new class label is predicted and assigned to each piece of textual data that can be either a document or a sentence. Any text classification task mostly involves following tasks - data processing, features extraction, model building & evaluation. There are 4 varied text data sets - *authors, products, 4dim, urdu* that need to be classified. For this, efforts have been made to go slightly beyond the usual *"it depends"* answer to **"which model to use for classification?"**.There are many state of the art text classifiers like Decision Trees, SVM, Random Forest, Perceptron, Naive Bayes, Logistic regression etc. This report is concentrated on the last three classification models for classifying the above mentioned data sets. It also gives a comparative study of performance of models when they are coded from scratch 'vs' when they are used directly from the state of the art libraries like *sklearn*. It is *important to note* that the features have been kept constant for these models for a fair comparison of the models with a fixed set of features.

2 Data Description

There are 4 different data sets namely - *'authors', 'products', '4dim', 'urdu'*. Each data set consists of lines that's text and label separated by a tab. Data sets are of varying sizes as shown in plots below: A brief description of the data:

Figure 1: All data sets used



- authors: Short lines of English poetry by either Emily Bronte (b) or William Shakespeare (s).# 3665
- products: Very variable length lines of various kinds of English product reviews that

are either positive(pos) or negative (neg).# 32,592

- 4dim: English reviews of variable length that are positive or negative and truthful or deceptive (pos.tru,pos.dec, neg.tru, neg.dec).# 1560

- urdu: Urdu BBC news articles either about Pakistan (pakistan) or the rest of the world (world).# 220

This paper, for the sake of simplicity, focuses on just one data -'authors'. Machine Learning models need to be tested on unseen data to gauge their true on-field performance. Also, we need a subset of the data to be used as development set in order to tune model parameters. So, I divided each of these data into 3 parts in the 80:10:10 ratio for *training data, development data and test data* respectively.

3 Fetaure Extraxtion

To be able to make sense of the data, all of data sets are engineered for features. Features are concise of describing the textual data numerically.

3.1 Data Pre-Processing

Processing of the data to plays a vital role in understanding the data as well as making the model more robust. Text data might have erroneous characters or words or sometimes even sentences that don't make sense and should be removed from the data for robustness.For example characters like '‹', ';', '()', '' etc. can be replace by empty space if they don't add any understanding to the corpus. Removal of stop words from the vocabulary also helps in catering to the long tail issue of the textual data. For *'authors'* data set, I looked at the histogram of words before the pre-processing and it very clearly had a long tail issue. This is in accordance with the *Zipf's law* for the any corpus of natural language. As a pre-processing step apart from removal of space, stop-words, punctuation's etc., I addressed the long tail issue of the data by removing top words constituting the top 10 percent of the entire word frequency. This pre-process step ensures our classification algorithm is not misguided by the artifacts of the grammar of a particular language (In general, an English corpus will see more determiners/pronouns/prepositions than actual verbs). The post processing data doesn't resemble normal distribution exactly but is much better in avoiding the long tail issue as shown earlier.

Figure 2: Distribution of words frequencies before pre-processing in 'authors'

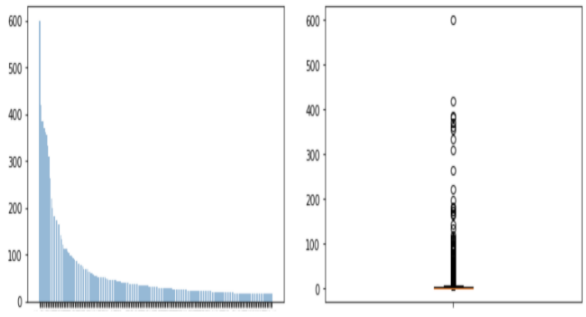
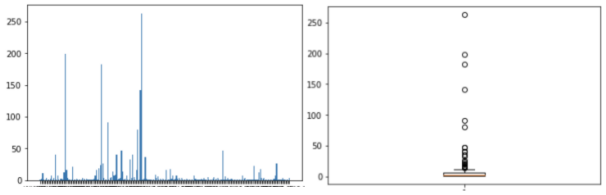


Figure 3: Distribution of words frequencies post pre-processing in 'authors'



3.2 Feature Engineering

Features are extracted from the textual data to capture the quirks of data set. Data domain expertise plays a very vital role in the feature identification. Features should be such that when combined together should be able to explain the data set well. The pipeline input consists of raw data set D = [X1, X2, ...Xn]. We identified a few basic features for 'authors' data after processing each X. *Note that in the interest of time, same features have been used across data sets*. These are:

- Length of text
- Count of all words, Count of unique words
- Average word length used, Maximum word length used
- Uppercase word counts, Title word counts, Punctuation counts etc

These features are very basic, more advanced features like POS tags frequency counts, n-gram based features, term frequency inverse document frequency(TF-IDF), word or sentence embedding etc. could result have given even better results.

4 Modeling

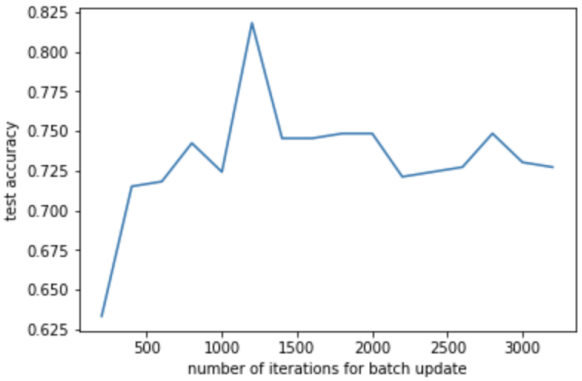
To better understand the modeling comparison and leanings mentioned in below sections, here are a few **assumptions that should be kept in mind**:

- Features had been generated only for one data set 'authors' for one model 'perceptron'. But same features were used across all data sets and models(except NB BOW and NB TF-IDF). It's not generalised and could lead to poor accuracy.
- The intent of the exercise was to implement models from scratch and understand the impact of various features, pre-processing on the model
- The goal was **not** to optimize the accuracy of the model but to understand why is the accuracy low and what can be done to boost it, I have identified enhancements and tried implementing them through pre-built functionalities of nltk, sklearn etc.

4.1 Perceptron

Perceptron model is a linear classifier that can learn from its mistakes and also doesn't have the feature independence assumption of Naive Bayes. A perceptron model starts with a random weight to features and updates its weights based on every error in the prediction.Every time, the model makes a mistake, the weight vector is shifted more towards the optimal weight vector. There is no reward if the prediction is right. I implemented *batch* update perceptron model. Feature set used for the model was the same as mentioned above. With 1200 iterations, I got an accuracy of **81.3%** on 'authors' data. I used the same feature set to generate the score of 'sklearn Perceptron' model- **78.2%**. This clearly indicates that my Perceptron model implementation is at par or even better with the sklearn's implemenation. This is because sklearn has auto assigned the maximum number of iterations to 1000 giving accuracy of 78.2% while I got an accuracy of 81.3% for number of iterations for batch update being 1200. Although, this accuracy is test accuracy but this might be overfitting the data given the dimension of training data is just 2968. It was observed that if we set the number of iterations to be really low or really high, the test accuracy of the model goes down because of the problem of underfitting and overfitting respectively(the highest accuracy is in the middle, as can be seen in Fig 4). Data is underfitted with Perceptron for number of iterations less than 300. Test accuracy gradually increases initially, shoots up to 82% and then again recedes with increase in number of iterations. It's clear from the plot that the stable range of test accuracy lies around 76% and should be preferred to avoid overfitting or underfitting issues. As the Perceptron model is able to fit the data really well, 'authors' data must be linearly separable to a large extent.**Future work on perceptron:** study the difference in the linear separability of data sets.

Figure 4: Change in test accuracy with number of iterations used for 'authors' data



4.2 Naive Bayes

I implemented the Naive Bayes model first with a few basic features that I finalised above for 'authors' data set. **First approach:** I used a version of the algorithm that supports numeric attributes and assumes the values of each numerical attribute are normally distributed. This is a strong assumption, but I wanted to check the result. I calculated summary stats(mean and standard deviation) of each numerical feature by class values.We can use a Gaussian function to estimate the probability of a given attribute value for a class, given the known mean and standard deviation for the attribute estimated from the training data.Now that we can calculate the probability of an attribute belonging to a class, we can combine the probabilities of all of the attribute values for a data instance and come up with a probability of the entire data instance belonging to the class. And the class that has the highest probability for the given data point is actual prediction.This of course, gave me a poor accuracy of **53%** on the validation set. Running 'MultinomialNB' model from **sklearn**,the accuracy was **59.8%** with the same feature set as used

for my hand-coded model. This slight difference was expected given the lack of efficient processing in my model creation.So, overall my hand coded model was working fairly good when compared to sklearn model with same features. **Second approach:** Next, I tried Bag of Words approach to be able to classify the data.In bag of words, a sentence is considered to be made of words and the probability of sentence given a class is given as product of probability of each word given the class. This is fit into the Bayes rule to get the most probable class for a given set of words forming a sentence.I am using 'add 1' smoothing to handle for unseen words in the data. The accuracy obtained with this model was **56.5%**. The poor performance with the Bag of Words model is majorly contributed by poor pre-processing of the data. After investigating into the incorrect classifications I have found out following **problems** that needed to be improved:

- there are multiple phrases connected by '-' and characters other than ' ' which could not be split into words during pre-processing
- the removal of just the top 10% of words from the corpus is not completely handling the long tail problem
- counting just the frequency of occurrence of words has the problem of not generalizing importance of the word for the sentence. A more frequent word occurring across sentences should have lower weight in identifying a specific sentence
- **Naive Bayes model works better** for large data sets like 'products' where it can learn variations

Third approach: Next, I wanted to use **better processing of the data**. This can be very easily achieved functionalities of NLTK library like 'tokenizer'. Words with very high frequencies were messing with the prediction although they were not the unique identifier of the sentence. So, I also wanted to incorporate the concept of weight to words in identifying a sentence based on Term Frequency and Inverse Document Frequency(TF-IDF). The document in our case actually represents sentences.*I have implemented TF-IDF Naive Bayes model too*. But the **problem** was the training computational considerations, my TF-IDF implementation considered only top 200 words for the most frequent vocabulary which is **not** reflective of the actual scenario and hence accuracy is low. In order to quickly check this hypothesis, I used the 'sklearn' module to implement this considering all words for the TF-IDF and the accuracy achieved was **82.6%**.

4.3 Logistic Regression

Unlike Perceptron, Logistic regression updates the weight vector of features using probability. There are many possible hyper-planes that can separate data and perceptron stops when it finds any of these hyper-planes. The **hyper-plane found by perceptron might not be the optimal** hyper-plane. That's why Logistic Regression is preferred since it gives the most optimal hyper-plane separating the data. Logistic regression minimizes the logistic loss. It updates the weight with every prediction in terms of reward or punishment. I implemented the logistic regression model using the same set of features as created for 'authors' for Perceptron model. The accuracy of my logistic regression model was **74.8%** with number of iterations for batch update being 1500. I compared it with the accuracy obtained my implementing sklearn's 'LogisticRegression' which was **78.2%** with the same feature set and number of iterations. The loss is really high in the beginning of the iteration and gradually decreases with the progress in the epoch and then stabilises towards the end of the epoch. Similar to perceptron, the accuracy is low with the lower number of epochs, increases with increase in epochs. As the number of epochs go beyond the size of the data itself, it's highly overfitted and the accuracy is not entirely trustworthy.

Figure 5: Change in test accuracy with number of iterations in Logistic Regression used for 'authors' data